# Implementing a Distributed Real-Time Database Manager

Sang H. Son, Marc S. Poris, and Carmen C. Iannacone

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
USA

## ABSTRACT

Conventional database systems are typically not used in time-critical applications due to poor performance and lack of predictability. Compared with traditional databases, database systems for such applications have the distinct feature that they must satisfy timing constraints associated with transactions. Transactions in real-time database systems should be scheduled considering both data consistency and timing constraints. In addition, a real-time database system must adapt to changes in the operating environment and guarantee the completion of critical transactions. In this paper we present our experience in integrating a relational database manager with a real-time operating system kernel. Since a database system must operate in the context of available operating system services, an environment for database systems development must provide facilities to support operating system functions and integrate them with database systems for experimentation. We choose the ARTS real-time operating system kernel, which provides a predictable and reliable distributed real-time computing environment.

## 1. Introduction

Real-time computing is an open research area [Stan88]. The growing importance of real-time computing in a large number of applications, such as aerospace and defense systems, industrial automation and robotics, and nuclear power plants, has resulted in an increased research effort in this area. In recent workshops sponsored by the Office of Naval Research [IEEE90, ONR89], researchers pointed to the need for basic research in database systems that satisfy timing constraints in collecting, updating, and retrieving shared data, since traditional data models and databases are not adequate for real-time systems. Very few conventional database systems allow users to specify or ensure timing constraints. Interest in this new application domain is also growing in the database community. Recently, a number of research results have appeared in the literature [Abb88, Abb89, Buc89, Kor90, Lin89, Lin90, Raj89, Sha88, Sha90, Son88b, Son89, Son90].

Time is the key factor to be considered in real-time database systems, and the correctness of the system depends not only on the logical results but also on the time within which the results are produced. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines, satisfying not only database consistency constraints but also timing constraints.

Conventional database systems are typically not used in real-time applications due to the inadequacies of poor performance and lack of predictability. Current database systems do not schedule their transactions to meet response requirements and they commonly lock data tables to assure only the consistency of the database. Locks and time-driven scheduling are basically incompatible, resulting in response requirement failures when low priority transactions block higher priority transactions. New techniques are required to manage the consistency of real-time databases, and they should be compatible with time-driven scheduling and meet both the required system response predictability and temporal consistency.

To address the inadequacies of current database systems, the transaction scheduler needs to be able to take advantage of the semantic and timing information associated with data objects and transactions. A model of real-time transactions needs to be developed which characterizes distinctive features of real-time databases that can contribute to the improved responsiveness of the system. The semantic information of the

This work was supported in part by ONR under contract N00014-88-K-0245, by CIT contract # CIT-INF-90-011, and by IBM Federal Systems Division.

*DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91*
Ed. A. Makinouchi
©World Scientific Publishing Co.

transactions investigated in the modeling study can be used to develop efficient transaction schedulers [Son88, Son90b].

Satisfying timing constraints while preserving data consistency requires the concurrency control protocol to accommodate timeliness of transactions as well as data consistency requirements. In real-time database systems, timeliness of a transaction is usually combined with its criticalness to take the form of the priority of the transaction. Therefore, proper management of priorities and conflict resolution in real-time transaction scheduling are essential for predictability and responsiveness of real-time database systems.

A database system must operate in the context of available operating system services, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. As pointed out by Stonebraker, operating system services in many systems are not appropriate for support of database functions [Ston81]. In many areas, such as buffer management, recovery, and consistency control, operating system facilities have to be duplicated by database systems because they are too slow or inappropriate. An environment for database systems development must, therefore, provide facilities to support operating system functions and integrate them with database systems for experimentation.

The ARTS real-time operating system kernel, being developed at Carnegie-Mellon University, attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time database system [Tok89]. The ARTS system, which provides support for programs written in C and C++, implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing as well as supporting lightweight tasks. All of these features are important when considering a real-time database.

We have been developing a new relational database manager for distributed real-time systems. We have used the relational database technology since it provides the most flexible means of accessing distributed data. In this paper we address the issues associated with developing a real-time database manager, describe the ARTS real-time operating system kernel, and present our distributed real-time database manager with performance results from a series of experiments for evaluating design choices.

## 2. The ARTS Real-Time OS Kernel

Research in the area of distributed, real-time operating systems indicates that most are designed for a specific need, and as such are different to build, maintain, and modify; in addition, they do not afford the capability of predicting runtime behavior during application design. In fact, few non-real-time operating systems provide a functionally complete set of general purpose, real-time task and time management functions, despite the fact that the user community is expressing the desire for increasingly complex applications of this type. Since the success of applications in real-time computing is primarily contingent on a system's temporal functionality, what is needed is an environment wherein the system engineer can analyze and predict, during the design stage, whether the given real-time tasks having various types of system and task interactions (i.e. memory allocation/deallocation, message communications, I/O interactions, etc.) can meet their timing requirements.

In an attempt to provide such functionality, the ARTS provides the process and data encapsulation that other distributed, object-oriented operating systems do, while at the same time including elements of temporal significance to the services it provides. This integration of data, thread and concurrency control greatly facilitates real-time schedulability analysis. The ARTS can support both hard and soft real-time tasks as well as periodic and sporadic ones [Tok89].

To support time-critical operations, the ARTS programming language interface allows designers to specify timing requirements and the chosen communication structure so that they are visible at both the language and system level; this allows the system-wide ARTS environment to make scheduling decisions based on both temporal constraints and priority. The Integrated Time-Driven Scheduler (ITDS) model of the ARTS is more effective than the common priority-based preemptive scheduling of many real-time systems. Such simple schedulers become confused during heavy system loads when they cannot decide which tasks are important and should be completed and which tasks should be aborted, causing unpredictability in the applications. The ITDS model however, employs a time-varying "value function" which specifies both a task's time criticality and semantic importance simultaneously. A hard real-time task can be characterized by a step function where the discontinuity occurs at the deadline, while soft real-time tasks are described by continuous (linear or non-linear) decreasing function after its critical time. In addition, ARTS' designers have separated the policy and mechanism layers, so that users can implement new scheduling policies with a minimum of effort, even dynamically changing the policy during runtime.

The issue of priority inversion is crucial to providing semantically correct system behavior in addition to addressing temporal concerns. Priority inversion occurs when a high priority activity waits for a lower

priority activity to complete. Resource sharing and communication among the executing tasks can lead to priority inversion if the operating system does not manage the available resource set properly. Significant research in the construction of ARTS was done to avoid priority inversion among concurrently executing tasks; in the processor scheduling domain, low priority servers which provide service to clients of all priorities are susceptible to inversion. For example, when a low priority request is being serviced, a high priority task requests the same service; since the server's computation is non-preemptable, the high priority request waits. Any task of higher priority than the server may preempt the server itself, however, so if a medium priority task arrives it preempts the server indefinitely, causing the high priority job to be lost in the shuffle. The ARTS employs a priority inheritance mechanism to propagate information about a single computation which crosses task boundaries. That is, if a server task accepts the request of a client, the server inherits the priority of the client. Furthermore, the server should also inherit the priority of the highest priority task waiting for the service [Tok89].

The notion of time encapsulation cannot be divorced from the basic structure of ARTS, in which every computational entity is represented as an object, called an *artobject*. An artobject is defined as either a passive or an active object. In a passive object, there is no explicit declaration of a thread which accepts incoming invocation requests while an active object contains one or more threads defined by the user. In an active object, its designer is responsible for providing concurrency control among coexecuting operations. When a new instance of an active object is created, its root thread will be created and run immediately. A thread can create threads within its object.

The ARTS kernel supports the notion of real-time objects and real-time threads. A real-time object is defined with a "time fence," a timer associated with the thread which ensures that the remaining slack time is larger than the worst case execution time for the operation. A real-time thread can have a value function and timing constraints related to its execution period, worst case execution time, phase, and delay value. When an operation with a time fence is invoked, the operation will be executed (or accepted) if there is enough remaining computation time against the specified worst case execution time of the operation for the caller. Otherwise, it will be aborted as a time fence error. The objective of this extension to a normal object paradigm is to prevent timing errors from crossing task or module boundaries (as often happens in traditional real-time systems which use a cyclic executive) and bind the timing error at every object invocation.

Some ARTS objects offer a single operation implemented by a single thread (SOST) or multiple operations implemented by a single thread (MOST). Other objects offer a single operation implemented by multiple threads (SOMT) or multiple operations implemented by multiple threads (MOMT). Some differences exist in terms of priority inheritance depending on the object implementation. Although beyond the scope of this introduction, the CMU researchers utilize two distinct methods which may be used to assign incoming invocations to object threads depending on priority. One method dynamically changes the priorities of the object threads based on the priority of the invocation being serviced. The other method utilizes a pool of prioritized object threads and assigns the incoming invocation requests to a thread of the appropriate priority.

On top of the ARTS foundation we have built a relational database manager using message passing primitives and employing the client/server paradigm. The result, RTDB, currently consists of a multi-threaded server which accepts requests of several clients. Based on the temporal urgency of the request, the server determines whether it can commit the transaction or it has to reject it.

### 3. Design and Implementation of the RTDB

The RTDB is a relational database manager written in C designed to run on ARTS as well as on UNIX. It offers not only a functionally complete set of relational operators-- such as join, projection, selection, union, and set difference-- but also such other necessary operators as create, insert, update, delete, rename, compress, sort, extract, import, export, and print. These operators give the user a good amount of relational power and convenience on managing the database.

Initially the RTDB runs on UNIX, where all relations are stored as files on disk. It was designed as a single-user program, and hence the code was not necessarily re-entrant. Many changes had to be made to have the RTDB run on the ARTS in a server-client mode with the server being multiply threaded so that it can accept requests from multiple clients possibly on different machines. For example, to support multiple threads the code had to be made re-entrant. In addition to the user interface in the form of an interactive command parser, the RTDB server provides a communication interface using packets generated by client objects.

We have developed two different kinds of clients for the RTDB. One is an interactive command parser/request generator that makes requests to the server on behalf of the user. This client looks and behaves just like the single-user database manager running on UNIX. It is possible to run the client without knowing that any interaction between server and client

is occurring. The other client is a transaction generating client, representing a real-time process that needs to make database access requests.

### 3.1. Single-threaded Server

The RTDB server is the heart of the database management system. It is responsible for creating and storing the relations, receiving and acting on requests from multiple clients, and returning desired information to the clients. The server contains an infinite loop that accepts high-level database requests (e.g., print, insert, join) from any client. The requests come in as packets. The RTDB provides two different types of packets: *call* packets and *return* packets. The call packet, created by the client, contains all the information that the server needs to carry out the desired database access operation. Since different commands require different information, the call packet has a variant field containing different information for each command. When the server completes the processing of the request, it returns a packet to the client with the information requested. This packet is called a return packet. The return packet is created by the server and also has a variant field that carries command specific information.

The communication between the server and clients is performed by the ARTS communication primitives: Request, Accept, and Reply. The communication is synchronous; when a client issues a Request, it is blocked until the server Accepts and Replies to the message. This may cause some problems, especially in a real-time environment, for two reasons: priority inversion and data sharing.

The ARTS kernel (and thus the RTDB system) supports eight message priorities. When the server Accepts a message it gets the highest priority message from the message queue. The server then executes the request to completion and Replies to the client with its information. If any higher priority messages arrive in the meantime they end up waiting on the lower priority work to finish, resulting in a priority inversion.

The server Replies to a client without completing a request when it needs to return more information than can fit in a single packet. In such a case, the client must make continuation requests to the server until it gets all the information requested. It is in this way that more than one client can have outstanding requests being worked on at the same time, because the server can accept other requests while the client is processing that part of the information that the server returned. The problem here is that the relations opened by an unfinished request remain open until the original request is *completely* finished. This can cause other requests to be blocked while waiting on the file(s) in use. The solution to this problem is to use a lock table that keeps track of which relations are in use at any given time. If a request for file A comes in while file A is being used by another ongoing command, then the new request must be put on an internal queue until all the files it needs are available.

The internal queue must also be a priority queue and it must be checked before accepting any new requests. The problem with this is that there may be an incoming request of higher priority than the highest blocked request. Our solution is to dequeue both an incoming message (if there is one) and the highest priority blocked message that has become unblocked. The priorities of these requests are compared and the request of higher priority is executed, while putting the lower priority request on the internal queue.

### 3.2. Multi-threaded Server

We have investigated better solutions to the problems discussed above. One of them is to have a server with multiple threads. The ARTS kernel supports lightweight processes which means that a single object can have many active control threads at the same time. This is implemented using a shared memory address space for the threads. A server that takes advantage of this feature can be designed in a number of different ways. There can be threads that accept only requests for a certain type of operation, threads that accept requests for any operation of a certain priority, or threads that accept any operation of any priority (which would be just like having more than one complete server). We have implemented the third choice.

The server object defines three threads. The root thread is automatically executed by ARTS upon invocation of the server. The server activates one or more worker threads, and activates a backup thread which is responsible for periodically backing up the relations that reside only in main memory.

The root thread of the server is responsible for binding the server's name in the ARTS name server so that the clients can find it and send requests. It is also responsible for reading the relations into memory, initializing the lock table, initializing the blocked request queue, instantiating the backup thread, and instantiating the server worker threads.

The worker thread of the RTDB server performs the client's request to access the database. It accepts requests, carries out the work that is requested, and replies back to the client. To maintain the consistency of the database, the RTDB server also need to handle conflicting requests properly. For example, some of the request has to be blocked since it needs to lock a relation that is already locked.

Whenever the server becomes free, it first checks the priority queue of blocked requests. If there are any requests in the block queue that can be unblocked, it

dequeues the highest priority request and processes it. If no request in the block queue is ready to be processed, the server worker Accepts an incoming request message. The problem with this approach is that a priority inversion may occur if the highest priority request in the block queue is of lower priority than the priority of incoming request. ARTS provides a CheckRequest primitive that checks to see if there are any incoming requests. However, using this primitive, a server worker cannot compare the priorities between the blocked requests and incoming requests. A more powerful primitive is necessary that can return the highest priority of incoming requests. A more desirable primitive would be one that allows the application program to specify a priority level in the Accept which indicates the minimum allowable priority request. For instance, if it is desirable to accept requests of at least priority 3, the server passes the priority 3 as a parameter to the primitive.

## 4. Server Thread Management

In the current RTDB implementation we employ a multiply threaded server. Efficient exploitation of the potential advantages of the multiply threaded server is our current research focus. *Mediating*, or scheduling the task workload among the active threads within the RTDB server object, is necessary to implement various algorithms which deal with semantic information provided by the clients and/or the task requests (i.e. temporal issues, priorities, etc.). In designing the mediator, we developed two models which merit further study.

The first model instantiates a new object specifically for this task. The CLIENT's request packets are accepted by the MEDIATOR object which examines and categorizes them based on the criteria used in the scheduling algorithm, and assigns the task to a worker thread in the SERVER object. The MEDIATOR needs to know some information currently within the sole domain of the SERVER object (i.e., the locktable status, viability of the individual server worker threads, etc.).

The second model constructs a mediation mechanism within the RTDB SERVER object, presumably as a separate, spawned thread of the srv_root thread. The inherent disadvantages of this design are the difficulty of adding and changing scheduling policies used in the mechanism without affecting the SERVER object code, and the decreased abstraction afforded the overall system by the previous model.

Currently we are examining issues associated with each model. For example, we need to decide which data structures belong to which object. Using the thread model of figure 2, should the lock table be removed from the SERVER OBJECT and placed in the MEDIATOR OBJECT? Also, what kind of handshaking procedure needs to be involved among the objects to insure data consistency while maximizing throughput? To answer these and similar implementation questions, we are examining recent research by others in this area, especially the work by the ARTS Project researchers at Carnegie-Mellon University. Lastly, determining the proper balance of control between ARTS primitives and RTDB explicit mediation will help us achieve the most beneficial symbiosis of the system's resources, the ultimate goal of our research. Implementing the mediator will not only require addressing this issue, but will also provide a mechanism for assessing the effects of our decisions.

## 5. Performance Evaluation

We have evaluated the current implementation of the RTDB server/client for various commands. The performance of the RTDB is measured by using the RTDB client object that generates transactions and sends them to the server. The granularity of measurements that we make is one full command. The time it takes to perform the command begins *before* the client performs the Request primitive and ends when the client has regained control *after* its Request has finished. During this time, the server performs Accept operation to receive the request, performs the requested work, and Reply results back to the client. Therefore, the measurement includes the interprocess communication time in addition to the command execution time.

The ARTS/RTDB supports three different methods to store and access data: UNIX file system manager, ram-disk, and memory resident files. Each method has its own advantages and drawbacks in terms of compatibility, performance, and recoverability. One of our objectives of the RTDB performance evaluation was to measure the performance gain that could be achieved by using different methods (i.e., using memory resident data as opposed to using disk resident data).

### 5.1. UNIX File System Manager

Because implementing file management in hard real-time systems is very difficult-- since there are many unpredictable delay factors involved in interacting with permanent disk storage-- ARTS does not support a file system as part of the operating system [Ash90]. In order to load objects or data onto ARTS, it is necessary to use its ufsm (UNIX File System Manager). The ufsm runs on a UNIX file server located on the same network as the ARTS machine. It is responsible for handling certain UNIX file system calls (e.g., read, write, open, etc). It is not necessary to change code written in C that runs on UNIX in order to get the file operations to work; if the ufsm is running on the file server, then the ARTS uses those files.

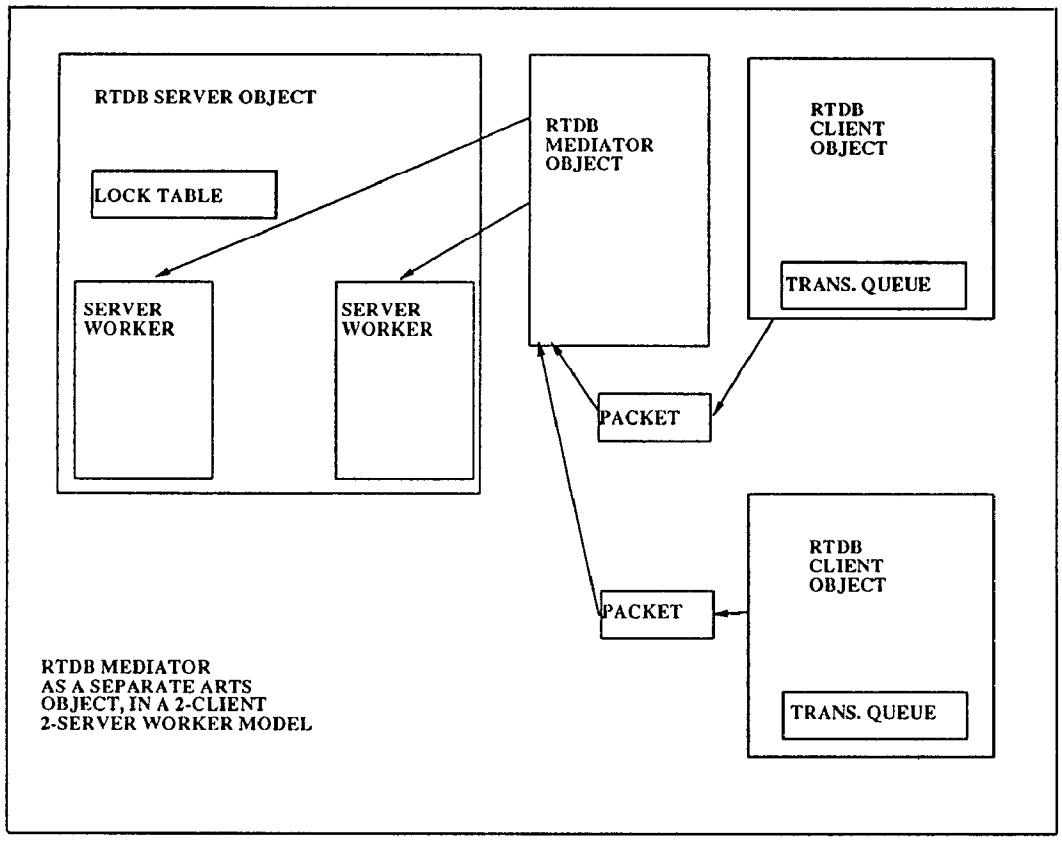



Figure 1. Mediator object model

The ufsm cannot make any real-time guarantees on how long each request will take, because UNIX makes no guarantees on when a process will be scheduled or how long it will take to read data from the disk. In our testbed environment, the ufsm runs on a high traffic file server (public Ethernet file server), which makes the RTDB system very slow. Clearly, using the ufsm for a real-time database is not very desirable in terms of the system responsiveness. However, it is compatible with the UNIX file system and it can provide the same level of recoverability as the UNIX file system.

### 5.2. Ram-disk

The ARTS supports a hard ram (hram) object with a UNIX compatible file system interface [Ash90]. The hram provides fast and predictable access to ram-disk files. By mounting the hram to a logical directory, it is not necessary to change any UNIX file system calls to access ram-disk data. The hram provides the RTDB a means to eliminate crossing the network every time file access is needed. In this way, the hram improves the performance and the system responsiveness tremendously. However, the hram still adds a level of I/O abstraction. With the hram, whenever a write or read is performed to a file, it is actually being performed on a certain location in the kernel's address space. If we remove this level of abstraction in the RTDB and manage the relations in the memory of the server, then we should expect a performance gain.

### 5.3. Memory Resident Data

The current implementation of the RTDB takes the advantage of memory resident relation files. As described in the previous section, the server reads the relations into memory before any database commands can be executed. The read and write operations that were previously handled by either the ufsm or the hram now access the server's virtual memory space. Since the hram is an object that receives and sends messages to the file system primitives, a whole level of I/O abstraction and the communication overhead can be removed by storing the relations in memory. This memory resident files, therefore, should improve performance over the hram. However, this method is feasible only for real-time databases with a limited number

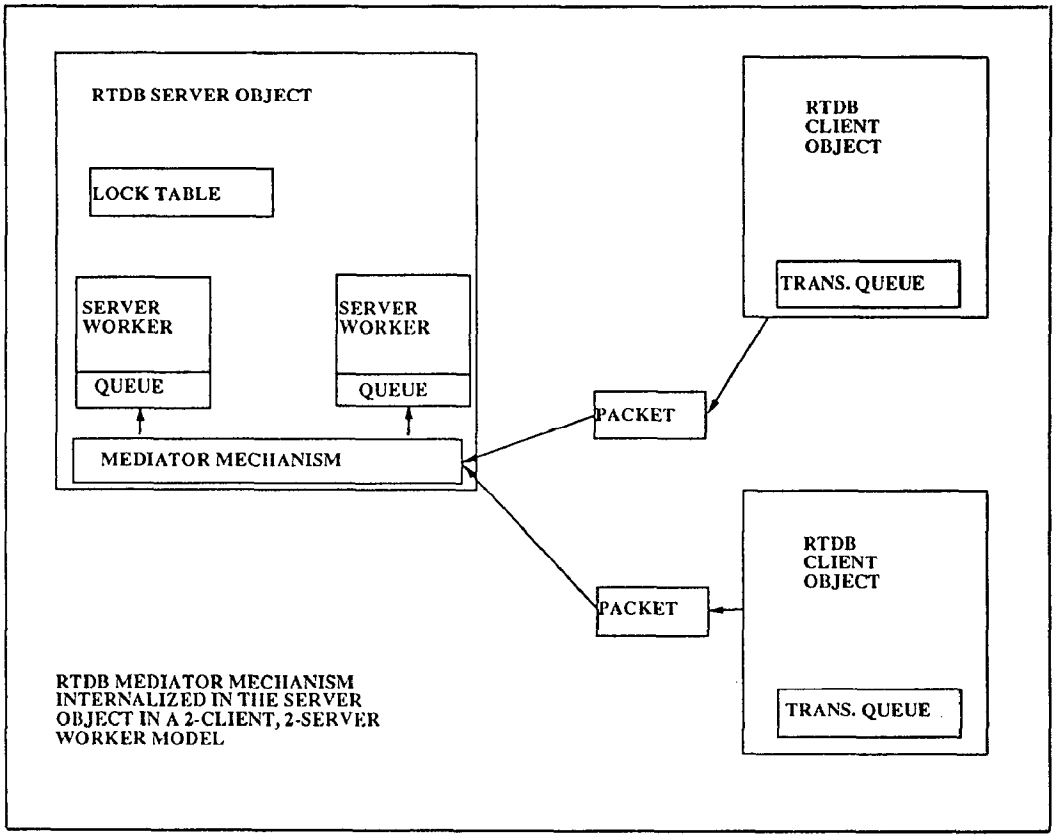



Figure 2. Mediator thread model

of relations of reasonable sizes.

### 5.4. Performance Results

We have run experiments for various commands on each access method, summarized in Table 1. One of the problems in the experimentation was that each run of a command can be measured only to the nearest 10 milliseconds, because the clock of the Sun workstation on which the ARTS/RTDB is running has a time granularity of 10 milliseconds. Since this is not fine grained enough for our experiments, it was necessary to repeat the commands numerous times in order to get accurate timing results. The data shown in Table 1 for one client represents the average of 1000 repetitions of each command. The data shown for two clients is somewhat inaccurate; commands are not repeated when multiple clients are used. However, the data represents the average of multiple runs of these commands.

The performance data in Table 1 illustrate the performance improvement that can be achieved by moving the relations from the UNIX file system to the ram-disk and then to the server's memory space. The commands we have tested are very simple ones, such as inserting one tuple, updating one tuple, and printing one tuple.

Note that this performance result is achieved without any optimization. Considering all the priority-based, time-driven scheduling capabilities provided by the ARTS kernel, in addition to optimization not yet implemented (e.g., buffering and data reallocation), this performance is very promising.

We also have studied the effects of conflicting requests on the response time by running the experiments for the PRINT command issued by two clients. We found, as expected, that the average response time increases almost linearly as the probability of conflicts increases. It has confirmed our conjecture that conflict resolution is one of the key components that dominates the responsiveness of relational databases. The problem here is that unless the system can predict when and how conflicts may occur, the system responsiveness can vary widely. One possible improvement of the database manager in this regard is to use better scheduling and conflict resolving protocols. Since not

all the transactions require strong consistency of data objects maintained by serializability, methods for selectively applying different consistency management techniques and hence guaranteeing timely responsiveness should be investigated and integrated with the current implementation of the relational database manager.

## 6. An Application of the RTDB

One of the applications of the RTDB is the Distributed Operating System Experiment (DOSE), as presented in [Butt90]. The goal of DOSE is to evaluate the feasibility of using a database kernel in embedded systems with requirements for high performance and real-time priority and predictability guarantees.

The DOSE application consists of data input, storage, display, and retrieval functions. These functions are implemented by four components: parser manager (PM), track report manager (TRM), graphics map client (GMC), and database monitor client (DMC). The PM receives tacking data from data terminals or communication links and converts them into a useful format such as floating point and signed integer numbers. The PM does not retain any incoming or outgoing information. The parsed data coming out from the PM are stored by the TRM. For each new incoming tracking data, a new data object is created. For high reliability, TRM maintains replicated data objects. The GMC enables the data to be mapped out and visualized on screen. It periodically checks with the TRM for the latest updates to be displayed. The DMC monitors the data objects in each replicated TRM database. Using frequent updates, it guarantees that data would remain consistent across the replicated TRM databases. Without DMC, the survivability and consistency of the system would be weakened.

| Command | Description |
|---|---|
| compress | moves the active tuples to the top of a relation by opening multiple "cursors" or tuple-pointers, and searching sequentially through the file, swapping tuples which have their *deleted* flag set with subsequently-appearing active ones |
| extract | returns the header information for a relation; this consists of field names, types, widths, and an estimate of the maximum tuple count. |
| insert_one | inserts a single tuple into a relation |
| print_one | retrieves a single tuple from a relation (does not include screen-writing I/O cost) |
| update_one | updates a single tuple in a relation with randomized data |

**ufsm**

| # of clients | compress | extract | insert_one | print_one | update_one |
|---|---|---|---|---|---|
| 1 | 1208 | 144 | 210 | 118 | 120 |
| 2 | 1710 | 523 | 380 | 508 | 228 |

**hram**

| # of clients | compress | extract | insert_one | print_one | update_one |
|---|---|---|---|---|---|
| 1 | 370 | 46 | 60 | 30 | 30 |
| 2 | 453 | 181 | 107 | 180 | 115 |

**memory**

| # of clients | compress | extract | insert_one | print_one | update_one |
|---|---|---|---|---|---|
| 1 | 115.4 | 24.1 | 6.5 | 4.9 | 5.3 |
| 2 | 194.8 | 57.3 | 33.6 | 35.0 | 32.8 |

Table 1. Selected RTDB commands and their response times (in milliseconds)

The scenario used with the DOSE application is an outer air battle scenario generated by IBGTT, the Interim Battle Group Tactical Trainer. The data generated by IBGTT consists of coordinate and motion data as well as general military classifications of tracked objects, called *platforms*. This data can be used to plot tactical information for a variety of situations, including personnel training programs, strategic simulations, and real-time military surveillance. Table 2 shows the attributes of data objects used in the DOSE application.

## 7. Conclusions

A real-time database manager is one of the critical components of real-time systems, in which tasks are associated with deadlines and a significant portion of data is highly perishable in the sense that it has value to the mission only if used quickly. To satisfy the timing requirements, transactions must be scheduled considering not only the consistency constraints but also their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire.

In this paper, we have presented an experimental relational database manager developed for distributed real-time systems. Since the characteristics of a real-time database manager are distinct from conventional database managers, there are different kinds of issues to be considered in developing a real-time database manager. For example, priority-based scheduling and memory resident data have been investigated in the development of the RTDB.

The RTDB is still in its development stage. The foundation now exists for a good real-time relational database manager, but there are many technical issues associated with real-time transaction management that need further investigation. For example, priority-based deadline-driven scheduling protocols and multi-threaded server design are two of such key issues. In many priority-based transaction scheduling protocols preemption is usually not allowed. To reduce the number of deadline-missing transactions, however, preemption may need to be considered. The preemption decision in a real-time database system must be made very carefully, and as pointed out in [Stan88], it should not necessarily be based only on relative deadlines. Since preemption implies not only that the work done by the preempted transaction be undone, but also that later on, if restarted, the RTDB must redo the work. The resultant delay and the wasted execution may cause one or both of these transactions, as well as other transactions, to miss their deadlines.

A multiply-threaded server can be designed in a number of ways. We have implemented the choice of multiple identical workers. Currently, we are investigating issues associated with implementing multi-threaded server in which a thread with priority $n$ would handle requests with message priority $n$. Several approaches to designing scheduling algorithms for real-time transactions have been proposed [Liu87,

| Attribute Name | Type | Meaning |
|---|---|---|
| trk_num | integer | track number |
| lat_track | real | latitude of track |
| long_track | real | longitude of track |
| bearing | real | bearing from data link reference point |
| dep_high | real | depth or height of platform |
| lat_dlrp | real | latitude of data link reference point |
| long_dlrp | real | longitude of data link reference point |
| platform_type | char | type of platform |
| cat | char | category of platform |
| time | integer | greenwich mean time |
| trkqa | integer | confidence of measurements |
| lat_tdir | char | latitude direction |
| long_tdir | char | longitude direction |
| course | real | bearing minus data link reference point |
| speed | real | speed of platform |
| range | real | range from data link reference point in nautical miles |
| nuclear | char | nuclear classification of platform |

Table 2. Data object attributes in DOSE application

Stan88, Abb89, Sha90, Son90c], but their performance in a realistic experimental environment has not been studied. The RTDB described in this paper with its multi-threaded server model, is an appropriate research vehicle for investigating such new techniques and scheduling algorithms for distributed real-time database systems.

## References


[Abb88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *VLDB Conference*, Sept. 1988, 1-12.

[Abb89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *VLDB Conference*, August 1989.

[Ash90] Ashida, N. and H. Tokuda, "Persistent Object/File Management for Hard Real-Time Operating Systems," *Techical Report*, Dept. of Computer Science, Carnegie Mellon University, 1990.

[Buc89] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Fifth Data Engineering Conference*, Feb. 1989, 470-480.

[Butt90] Butterbrodt, M. and J. Green, "DOSE: A Vital Link in the Development of a Real-Time Relational Database Environment," *Project Summary*, Naval Ocean Systems Center, Jan. 1990.

[IEEE90] *Seventh IEEE Workshop on Real-Time Operating Systems and Software*, University of Virginia, Charlottesville, Virginia, May 1990.

[Kor90] Korth, H., "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.

[Lin89] Lin, K., "Consistency Issues in Real-Time Database Systems," *Proc. 22nd Hawaii Intl. Conf. System Sciences*, Hawaii, Jan. 1989.

[Lin90] Lin, Y. and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990, to appear.

[ONR89] *ONR Annual Workshop on Foundations of Real-Time Computing*, White Oak, Maryland, Oct. 1989.

[Raj89] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Carnegie-Mellon University, August 1989.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Sha90] Sha, L., R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, to appear.

[Son88] Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology*, Vol. 30, September 1988, 443-449.

[Son88b] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son89] Son, S. H. and H. Kang, "Approaches to Design of Real-Time Database Systems," *International Symposium on Database Systems for Advanced Applications*, Seoul, Korea, April 1989, 274-281.

[Son90] Son, S. H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990, 124-131.

[Son90b] Son, S. H. and P. Wagle, "Scheduling Using Dynamic Priority in Real-Time Database Systems," *Technical Report TR-90-03*, Dept. of Computer Science, University of Virginia, March 1990.

[Son90c] Son, S. H. and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, 39-43.

[Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," *IEEE Computer 21*, 10, October 1988, 10-19.

[Ston81] Stonebraker, M., Operating System Support for Database Management, *Commun. of ACM 24*, 7 (July 1981), 412-418.

[Tok89] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23 (3), July 1989.